Understanding the Memory Consumption of the MiBench Embedded Benchmark

Antoine Blin, Cédric Courtaud, Julien Sopena, Julia Lawall, Gilles Muller

▶ **To cite this version:**

Antoine Blin, Cédric Courtaud, Julien Sopena, Julia Lawall, Gilles Muller. Understanding the Memory Consumption of the MiBench Embedded Benchmark. Netys, May 2016, Marakech, Morocco. hal-01349421

HAL Id: hal-01349421
https://inria.hal.science/hal-01349421

Submitted on 27 Jul 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Understanding the Memory Consumption of the MiBench Embedded Benchmark

Antoine Blin[1,2], Cédric Courtaud[1], Julien Sopena[1], Julia Lawall[1], and Gilles Muller[1]

[1] Sorbonne Universités, Inria, UPMC, LIP6
firstname.lastname@lip6.fr
[2] Renault S.A.S

Abstract. Complex embedded systems today commonly involve a mix of real-time and best-effort applications. The recent emergence of small low-cost commodity multi-core processors raises the possibility of running both kinds of applications on a single machine, with virtualization ensuring that the best-effort applications cannot steal CPU cycles from the real-time applications. Nevertheless, memory contention can introduce other sources of delay, that can lead to missed deadlines. In this paper, we analyze the sources of memory consumption for the real-time applications found in the MiBench embedded benchmark suite.

1 Introduction

In modern automobiles, computing is characterized by a mixture of real-time applications, such as management of the dashboard, the engine control, and best-effort applications, such as multimedia entertainment. Historically, multiple applications are integrated in a vehicle using a *federated architecture*: Every major function is implemented in a dedicated Electronic Control Unit (ECU) [23] that ensures fault isolation and error containment. This solution, however, doesn't scale in terms of costs, power consumption and network congestion when the number of functions increases. Recently, the AUTOSAR [16] consortium has been created to develop an *integrated architecture*, in which multiple functions share a single ECU. The AUTOSAR standard targets applications that control vehicle electrical systems and that are scheduled on a real-time operating system. Infotainment applications, however, typically target a Unix-like operating system, and thus still require the use of a federated architecture.

Recent experimental small uniform memory access commodity multicore systems provide a potential path towards a complete low-cost integrated architecture. Systems such as the Freescale SABRE Lite [1] offer sufficient CPU power to run multiple applications on a single low-cost ECU. Using *virtualized architectures* [12], multiple operating systems can be used without modification. Recent hypervisors targeting embedded systems, such as SeL4 [2] and PikeOS [5], make it possible in the context of the automotive domain to dedicate one or several cores to a class of applications, and thus provide CPU isolation.

CPU isolation, however, is not sufficient to ensure that real-time applications can meet their performance constraints. Indeed, some resources such as memory buses, memory controllers, and some caches remain shared across all cores. Therefore, it has been observed that the memory usage of applications running on one core may impact the execution time of applications running on the other cores [19, 21]. In recent work [13], we have shown that sharing these resources implies that the operations initiated on a best-effort core can affect the duration of real-time tasks running on other cores. In that work, we have developed an approach to address the memory induced slowdown that uses run-time monitoring to detect when the interference risks causing the real-time task to exceed its deadline beyond a threshold that is considered to be tolerable.

Assessing the benefit of this approach, and others like it that relate to the memory behavior of embedded systems [15, 19, 24, 26], requires appropriate benchmarks. We have used the MiBench benchmark suite [18], that has the goal of representing the spectrum of embedded applications used in industry. This benchmark suite is one of the few that targets the embedded computing domain, and is highly cited. Understanding the memory access behavior of the MiBench applications requires periodic profiling of the application execution. This profiling must be precise without excessively perturbing the application execution. Furthermore, for our experiments with MiBench to be meaningful, it must be the case that the memory access pattern of the MiBench applications is typical of that of embedded applications.

In this paper, we make the following contributions towards better understanding the memory behavior of the MiBench applications and making these applications better represent the memory behavior of embedded applications:

- We present the design of a memory access profiler that has little impact on the behavior of the profiled application, and we assess the various tradeoffs in this design.
- We use the profiler to detect spikes in the memory bandwidth usage of the 13 MiBench applications that we have previously found to be strongly affected by the memory usage of applications running on other cores.
- We use various techniques, including rewriting the application, overloading the C standard library, and changing the size of the file system buffers to isolate the reasons for the observed memory spikes. Based on the results, we classify the memory spikes into those that are derived from the behavior of the C standard library, of the operating system, and of the application.

The rest of this paper is organized as follows. In Section 2, we briefly present an overview of our hardware and of our software configuration. In Section 3, we present our profiler, evaluate its design decisions, and show the results of profiling the MiBench applications. In Section 4, we identify and classify the root causes of the spikes in memory usage observed in the MiBench applications. Finally, in Section 5, we present related work, and Section 6 concludes.

2 Platform

In this section, we first describe our hardware platform, and then we present MiBench and the software configuration used in our tests.

2.1 Hardware

We focus on embedded systems, as used in the automotive domain, which has strong hardware cost requirements. Therefore, for our tests we use the SABRE Lite board [22], a low-cost development platform designed for multimedia applications on the Android and Linux operating systems. A variant of this platform that has been adapted for the automotive domain is used by a large number of automotive manufacturers and suppliers. The processor of the SABRE Lite is an i.MX 6, which is based on a 1.2 GHz quad-core Cortex A9 MPCore [10]. Each core has a separate Level 1 (L1) 32-kilobyte 4-way set-associative cache for instructions and data [8]. All CPUs are connected to a single 1-megabyte 16-way set-associative L2 cache [9] that can be partitioned into multiples of the way size. Finally, the Multi Mode DRAM Controller (MMDC) manages access to one gigabyte of DDR3 RAM that can be used by all the cores [22].

The SABRE Lite board provides various hardware performance counters. Each core provides six configurable counters to gather statistics about the operation of the processor (number of cycles) and the memory system (L1 cache hits/misses) [7,8]. The MMDC has a profiling mechanism that gathers statistics (read/write bytes/access) about the global memory traffic on the platform.

2.2 Software stack

We use the applications of the MiBench [18] benchmark suite as real-time applications because this benchmark suite has been designed to be representative of embedded applications used in industry. This benchmark suite has been referenced almost 2700 times,[3] and thus is a reference benchmark in the academic domain. MiBench is composed of 35 embedded applications, mostly written in C, categorized into six subclasses: industrial control consumer devices, office automation, networking, security and telecommunications. Among these applications we omit 19 that contain x86 assembly code or that represent long-running or office applications. From the remaining applications, we select the 13 that are sensitive to the memory contention, as demonstrated by our previous work [13].

We run the MiBench applications on a Linux 3.0.35 kernel that has been ported by Freescale to the i.MX 6 architecture. All of the MiBench applications are compiled using GCC 4.9.1 with the option -O2. We use the 2.20 GNU C Library as the C standard library. On embedded platforms, the kinds of data inputs used by the Mibench applications are usually provided by external devices such as an on-board camera, network controller, or microphone that interact directly with the CPU, via DMA. To approximate this behavior without modifying the applications, we store the data inputs in an in-memory file system.

[3] Google Scholar, January 20, 2016

3 Memory Profiler

In this section, we present our memory profiler and show the memory profiles of the MiBench applications. We then study the benefits and costs of high resolution profiling.

3.1 Profiler overview

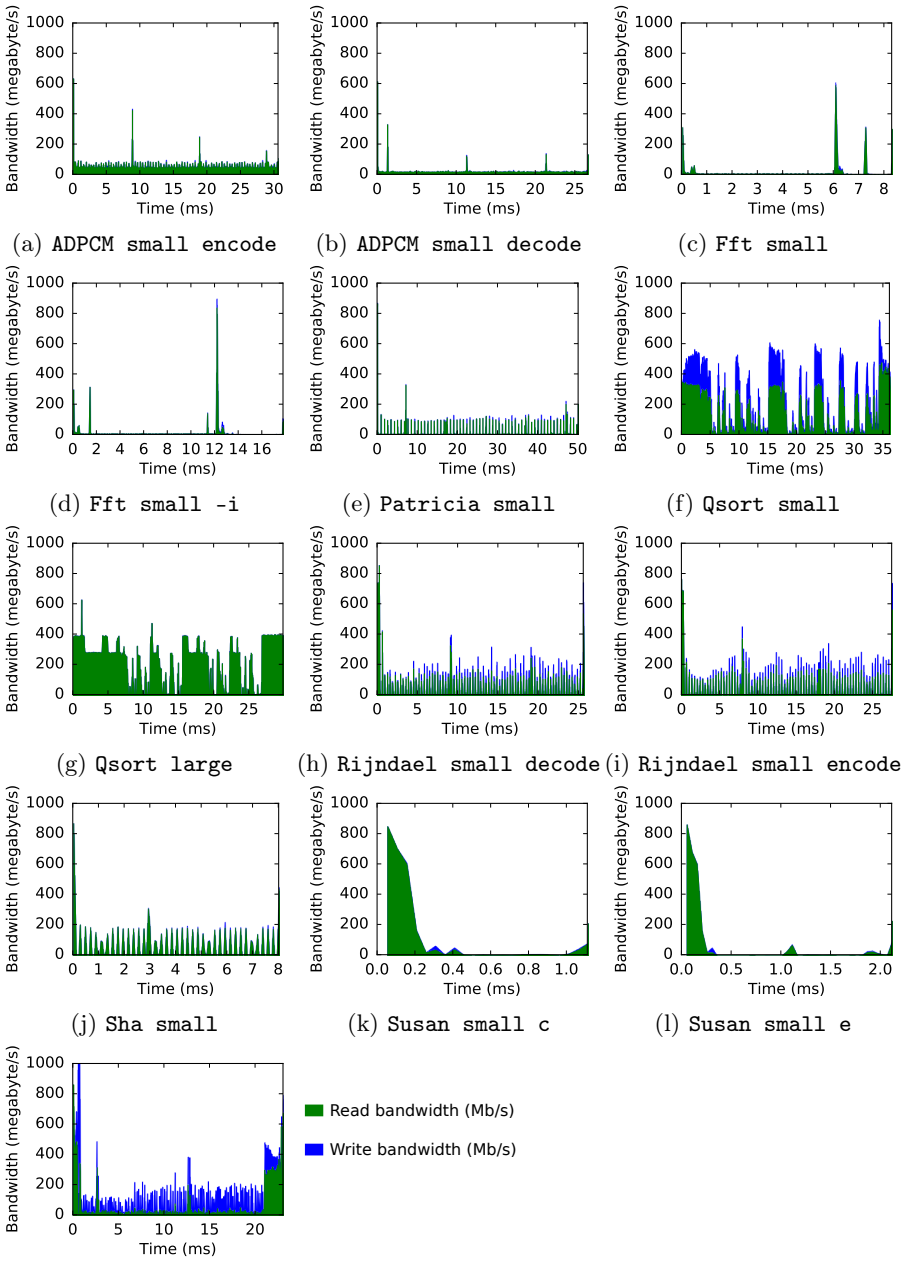
We have developed a memory profiling module for the Linux kernel that uses counters of the MMDC controller to measure global memory traffic. At the beginning of the profiling process we enable the cycle counter that counts the processor clock cycles. Our profiler then periodically samples several hardware performance counters to obtain information about the memory access behavior. Each sample contains the number of bytes read, the number bytes written and the value of the cycle counter. Samples are stored in memory during profiling and then are written to disk at the end of the application.

A challenge in designing a memory profiler is in how to enforce the sampling interval. One solution would be to use a timer interrupt. Peter et al. [25], however, have shown that timeouts set to a short intervals are frequently delivered a significant fraction of their duration after their expiry time. To allow profiling with intervals down to 1 us, we implement the sampling interval using a busy wait on a dedicated core. This approach allows calibrating the interval fairly precisely, but requires one core to be completely dedicated to profiling.

To prevent the profiler from interfering with the performance of the application, we pin the application to profile on one core (core 0), using the POSIX `sched_setaffinity` function, and pin the profiler to another (core 1). We disable the remaining cores. To avoid any preemption, we schedule the application to profile and the profiling thread using the `SCHED_FIFO` policy with highest priority, and we disable the Real Time throttling mechanism that gives control back to the operating system if a task has been scheduled for a time exceeding a specified delay. We further reduce interference between the application to profile and the profiler by partitioning the L2 cache between their different cores.

After the application execution has completed, the memory bandwidth is computed from the number of bytes read and written in each sample and the corresponding sample duration. The busy-wait delay approach induces small temporal variations between samples. We use the value of the cycle counter to make a temporal readjustment.

Figure 1 shows the resulting memory profiles of the 13 MiBench applications with a sampling interval of 50us. We observe that the profiling has no impact on the application running time. Based on the memory profiles, we classify the applications into two groups. The applications `ADPCM small encode`, `ADPCM small decode`, `Patricia small`, `Rijndael small decode`, `Rijndael small encode`, `Sha small` and `Susan large c` have regularly recurring spikes, while the remaining applications have a smoother memory profile.



(a) `ADPCM small encode` (b) `ADPCM small decode` (c) `Fft small`

(d) `Fft small -i` (e) `Patricia small` (f) `Qsort small`

(g) `Qsort large` (h) `Rijndael small decode` (i) `Rijndael small encode`

(j) `Sha small` (k) `Susan small c` (l) `Susan small e`

(m) `Susan large c`

Fig. 1: Memory profiles of selected MiBench applications (Sampling interval: 50 us)

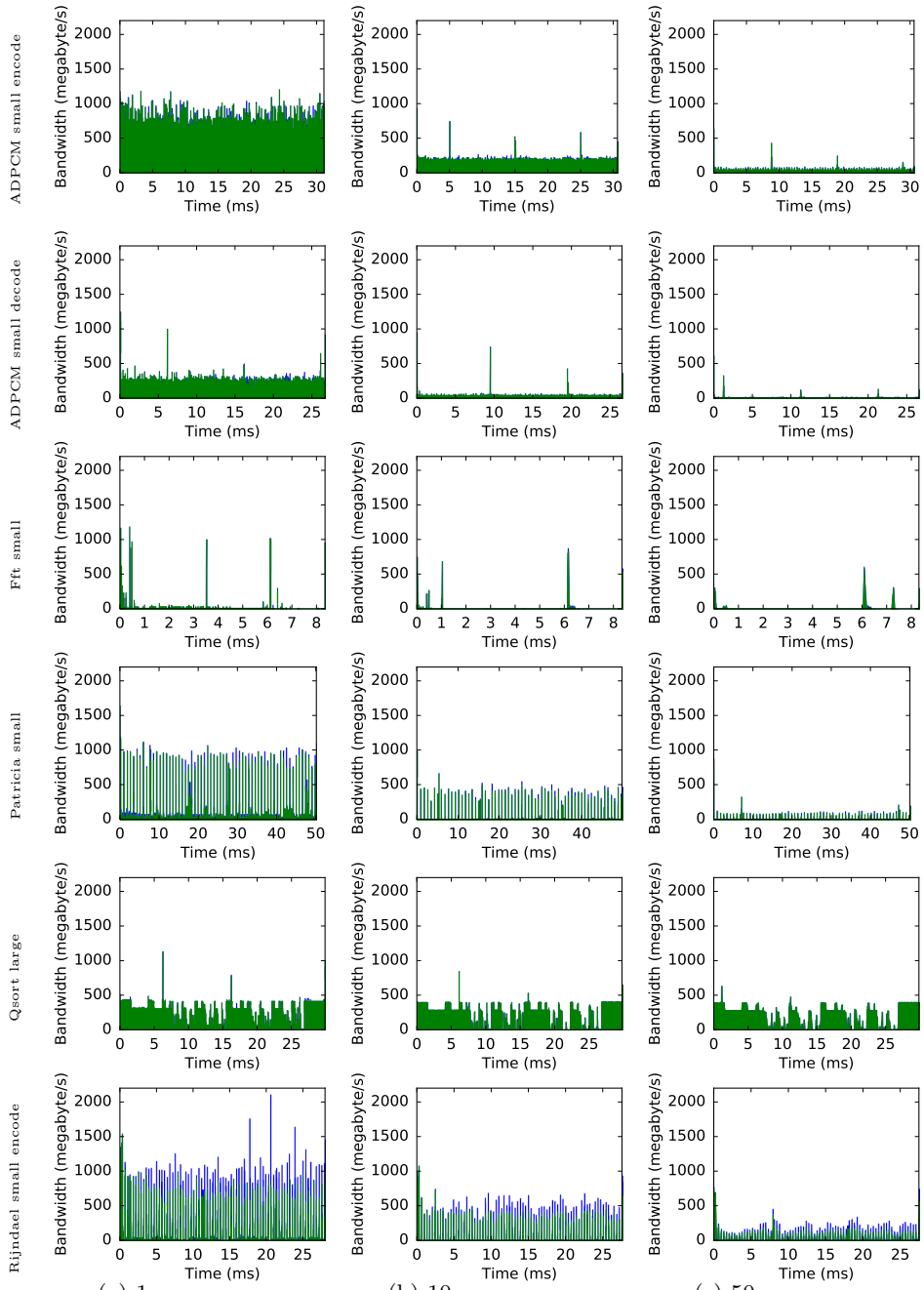
3.2 Profiler design choices

The profiler can be tuned with respect to the duration of the sampling interval. Indeed, the choice of sampling interval can have a signficant impact on the precision of profiling. A long interval smoothes the memory usage, because the profiled value represents an average over the sampled period, thus reducing the magnitude of spikes that have a duration shorter than the sampling interval. A short interval provides more accurate information about spikes, but may distort the resulting profile, because memory usage is measured globally, for the entire machine, and the profiler also uses memory, to record the profile information for each sample. Thus, the profiler can potentially introduce spikes or increase the size of existing ones. Furthermore, this added memory usage can potentially delay the memory accesses of the profiled application, and thus increase its running time. In this section, we explore these tradeoffs.

We first consider the bandwidth values observed with various sampling intervals. Figure 2 shows the memory profiles of selected MiBench applications with different sampling intervals. Of our original 13 applications, we have omitted `Fft small -i`, `qsort small` and `Rijndael small decode`, which have essentially the same profiles as `Fft small`, `qsort large` and `Rijndael small encode`, respectively. For the applications with memory usage spikes, decreasing the sampling interval from 10 us to 1 us or from 50 us to 10 us increases the height of the spikes by 2 or more times. For such applications we thus do not know the maximum bandwidth, as further decreasing the sampling interval may cause the observed spikes to increase even higher. For the applications without spikes, changing the sampling interval has little impact on the memory bandwidth pattern.

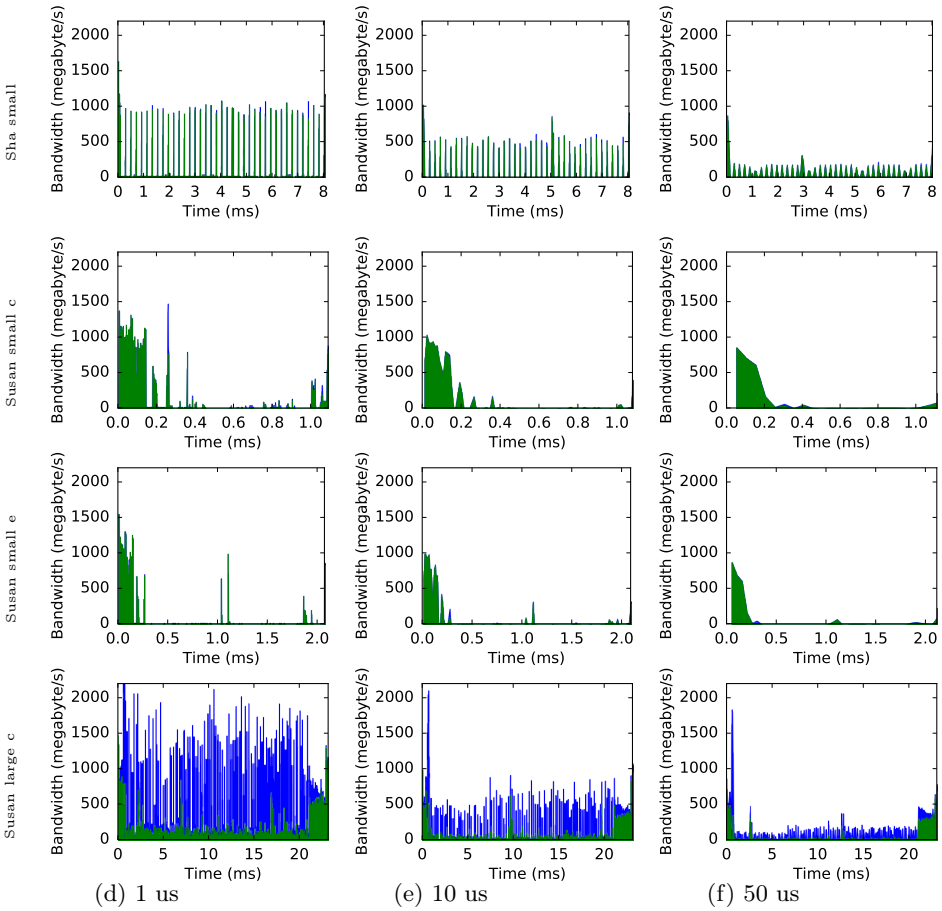
We next study the impact of the sampling interval on the overall memory usage of the system. For this, we develop and profile a very low memory footprint application, with the expectation that any observed memory usage in this case comes from the profiler itself. Our application simply increments a counter from zero to an upper bound. By varying the upper bound, we can control the application execution time. Figure 3 shows the overall memory bandwidth observed when profiling our low memory footprint application, with various execution times from 10 ms to 50 ms for the application and various sampling intervals from 50 us down to 1 us for the profiler. With a 50 us sampling interval, the bandwidth is always close to zero. With a 10 us sampling interval, the bandwidth is greater for short execution times than for long execution times, while with a 1 us sampling interval, the bandwidth generated by the profiler is more variable, with a bandwidth of up to 11 MB/s for a long running application. Nevertheless, even a memory bandwidth of 11 MB/s is negligible as compared to the memory bandwidths observed for most of the MiBench applications, and thus we consider that the memory bandwidth generated by the profiler is not an issue.

Finally, we did not observe significant differences in the application execution time with the various sampling intervals.



(a) 1 us (b) 10 us (c) 50 us

Fig. 2: Memory profiles of MiBench applications



(d) 1 us (e) 10 us (f) 50 us

Fig. 2: Memory profiles of MiBench applications, continued

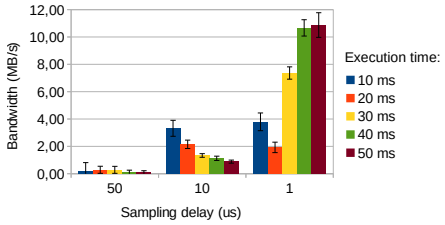


Fig. 3: Approximate memory bandwidth induced by the profiler

4 Origins of the Memory Spikes

For understanding memory behavior, spikes are problematic, because it is difficult to capture their real bandwidth due to their short duration. We now investigate the origins of these spikes. We first present a methodology to localize their origins. We then use various techniques, including rewriting the applica-

tion, overloading the C standard library, kernel modification, and changing the size of the file system buffers to identify the root causes of the memory spikes.

4.1 Methodology

To identify the origins of the memory spikes, we have developed a tagging mechanism. A tag is basically a set of instructions added in the application source code to obtain and record the value of the cycle counter. Using tags, we can relate a line of source code to the corresponding offset in the application memory profile. Using a large number of tags would substantially increase the application execution time. Therefore, we use tagging only for spike identification.

Our use of tags shows that the main causes of the spikes can be categorized into three groups: I/O functions, the operating system, and the application source code. In the rest of this section, we study each of these sources.

4.2 I/O functions

For each application that has spikes, Table 1 lists the I/O functions that are the sources of memory spikes. ADPCM uses the low level I/O functions `open`, `read`, and `close`, while the other applications use the buffered I/O functions `fopen`, `fread`, `fwrite`, `fgets` and `fclose`, which operate on streams.

Table 1: I/O function call sources of memory spikes

	open	read	close	fopen	fclose	fread	fwrite	fgetc	fgets
ADPCM small encode	x	x	x						
ADPCM small decode	x	x	x						
Patricia small				x	x				x
Rijndael decode				x	x	x	x		
Rijndael encode				x	x	x	x		
Sha small				x	x	x			
Susan small e				x	x	x		x	
Susan small c				x	x	x		x	
Susan large c				x	x	x		x	

Buffered I/O. To understand the impact of the buffered I/O functions we focus on `Rijndael small encode`, which performs both reads and writes. This application performs a computation on 16 byte blocks acquired using `fread`. Each computed block is then written to an output file using `fwrite`. We study the memory behavior of the reads and writes separately.

To study the memory traffic generated by `fread`, we simply comment out the `fwrite` calls, as the application's computation does not depend on them. The resulting memory profile (Figure 4b) shows a very low memory traffic mixed with regular spikes. Analysis of the tags shows that most of the spikes come from calls to `fread`. On average, there are 79 spikes greater than 600 MB/s per run, out of a total of 19489 calls to `fread`. Thus, on average, we have a memory spike every 246.7 `fread` calls. The size of the input file is 311,824 bytes. Thus, there is a memory spike every time a block of 3947 bytes has been read.

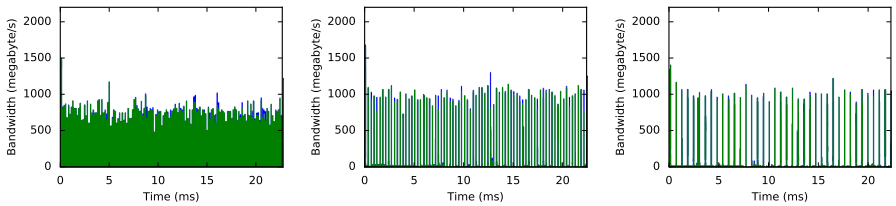
We hypothesize that the memory spikes come from the management of the internal input stream buffer. Using the `__fbufsize` function, we find that the default size of the input stream buffer is 4096 bytes. When `fread` is called, if the requested data are not present in the input stream buffer, `fread` refills the buffer by reading a new block of 4096 bytes from the file. Because the file is stored in a Temporary File System (TMPFS) mounted in memory, refilling the stream buffer involves a copy from the TMPFS to the input stream buffer, which generates a burst of memory traffic. All calls to `fread` also make a copy from the input stream buffer to the application buffer. This copy, however, does not generate any memory traffic, as both buffers are loaded in caches. These observations thus suggest that the memory spikes come from the refilling of the input stream buffer.

To test the hypothesis that the spikes are derived from filling the input stream buffer, we modified the size of this buffer using the `setvbuff` function. Figure 4 shows the memory profiles of `Rijndael small encode` with various input stream buffer sizes. Our hypothesis suggests that increasing the size of the stream buffer should reduce the number of times the buffer needs to be filled and should increase the height and duration of spikes, as filling a bigger buffer generates more memory traffic. Indeed, we see that increasing the size of the stream buffer reduces the number of memory spikes, slightly increases their height when moving from a 2 KB buffer to a 4 KB buffer, and increases their duration, according to the increase in the buffer size.

We next turn to writes. To analyse the memory traffic generated by `fwrite` we first override `fread` by a non-buffered read function, leaving the calls to `fwrite` commented. The resulting memory profile is shown in Figure 5, in which almost all of the large spikes have disappeared. The remaining spikes are due to operating system effects (Section 4.3). Then, we uncomment the call to `fwrite`. As shown in Figure 6b, the resulting memory profile contains, on average, 83 regular memory spikes per run that are greater than 300 MB/s. The output file size is 311,856 bytes. We thus have a memory spike every time a block of 3757 bytes has been written. We believe that the memory spikes issued from `fread` and `fwrite` functions have the same cause. We modified the size of the output stream buffer and again observe that the number of spikes decreases (Figure 6).

We performed the same experiments on all of the other applications that use buffered I/O functions and we observed the same behaviour.

Low-level I/O functions. ADPCM is a signal encoder/decoder application that performs computation on blocks acquired using `read`. The block size for `ADPCM small encode` is 2KB and the block size for `ADPCM small decode` is 500 bytes. For both applications, we observe that the number of spikes is exactly the same as the number of calls to `read`, which suggests that the spikes are due to the copy from TMPFS to the application buffer. To see the impact of the `read` calls, we reduce the application buffer size from 2KB bytes to 100 bytes. This change substantially increases the execution time of the application, but eliminates most of the memory spikes (Figure 7).



(a) 2 KB (b) 4 KB (Default) (c) 8 KB

Fig. 4: `Rijndael small encode` without writes and with various input stream buffer sizes

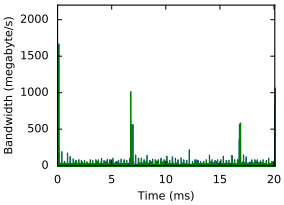
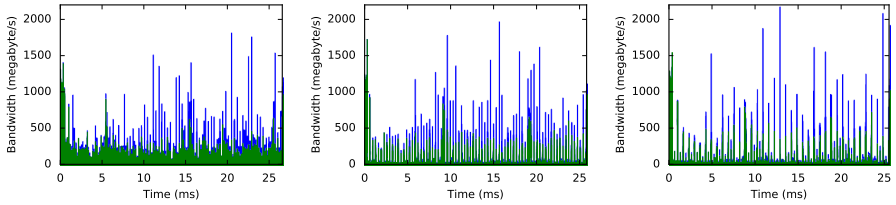
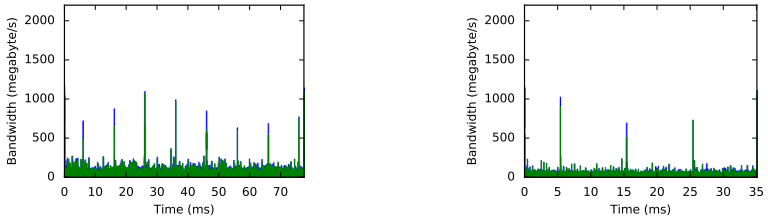


Fig. 5: `Rijndael small encode` without writes and with unbuffered reads



(a) 2 KB (b) 4 KB (Default) (c) 8 KB

Fig. 6: `Rijndael small encode` with unbuffered reads and with various output stream buffer sizes



(a) `ADPCM small encode` (b) `ADPCM small decode`

Fig. 7: `ADPCM small` with a small `read` buffer

In our view, the low-level I/O functions are representative of the memory accesses that could be done by an embedded application that would access data coming from an external device. The I/O operation itself can be done either by a

CPU copy loop or by DMA. On the other hand, buffered I/O functions generate additional spikes that are not suitable in an embedded context. We consider that this is a problem in the design of MiBench.

4.3 Operating system

Figures 5 and 7 show the memory profiles of applications that we have modified to eliminate the I/O induced memory spikes. These profiles, however, still contain a few spikes that occur at regular 10 ms intervals. We hypothesize that these spikes are due to the system timer of the operating system. The Linux timer frequency is defined at kernel compilation time by the configuration option `CONFIG_HZ`, which by default is set to 100 Hz, resulting in a timer interrupt every 10 ms. We generated the memory profile of our low memory footprint application (Section 3.2) with the timer frequency at 100 Hz and 50 Hz, and observed that the delays between spikes were around 10 us and 20 us, respectively, thus validating our hypothesis.

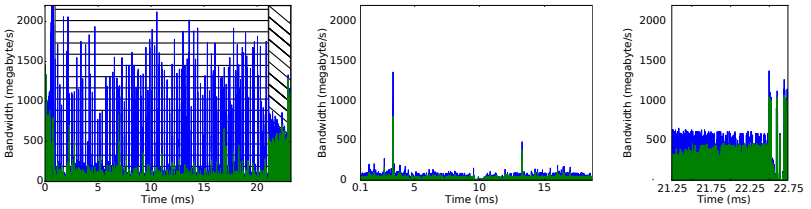
4.4 Applications

Most of the applications exhibiting spikes have a continuous behavior across the entire duration of the application. These applications follow the model of `Rijndael` and `ADPCM`, where the spikes are due to I/O functions and operating system effects. `Susan large c`, an image recognition package, on the other hand, has spikes, but has a more complex overall behavior. Specifically, the graphs for `Susan large c` in Figures 2 d-f show that the heights of the spikes are very variable, and for different spikes, the heights decrease at different rates as the sampling interval increases. To explore the reason for this behavior, we used the tagging mechanism, which revealed three different phases in the execution (Figure 8a).

In the first phase, lasting 0.1 ms, the memory spikes are generated by buffered I/O functions that load the image into a byte array.

In the second phase, lasting around 20 ms, we observe a very low read memory traffic mixed with regularly occurring write spikes of varying magnitudes. We used the tagging mechanism to identify the source code that generates the spikes. `Susan` uses a for-loop to iterate over the byte array. The loop body contains a complex condition which, when it succeeds, stores values in three integer arrays each having the same size as the image. These array stores are not sequential. Indeed, the average distance between two neighbouring stores is 287.5 bytes, with a standard deviation of 565.8, implying a huge variation. We hypothesize that the read traffic is derived from iterations over the byte array and the write spikes are derived from the stores into the integer arrays.

To test our hypothesis about the origin of the write spikes, we comment out the array writes. The written array values are not read during this phase, and thus removing the writes does not affect this phase's overall computation. Figure 8b shows the resulting memory profile of the second phase. Most of the write spikes disappear, thus validating our hypothesis.



(a) Phases (b) Phase 2 without writes (c) Phase 3 without writes

Fig. 8: `susan large -c`

In the third phase, we observe a high memory traffic of around 550 MB/s. `Susan` uses two nested loops to iterate over an integer array. These read accesses are not sequential. Every 384 iterations the application makes a jump of 288 iterations in the array. The inner loop body contains a complex condition which, when it succeeds, stores values in a structure array. We comment out these writes. Figure 8c shows the resulting memory profile of the third phase, in which the memory traffic is reduced to 500 MB/s. We further modified the code to remove the non-contiguous accesses and observed that the memory traffic decreased dramatically. We conclude that most of the memory traffic comes from the non-contiguous accesses.

5 Related Work

The main focus of our work is on memory profiling and on the behavior of benchmarks for embedded systems. Besides the original paper presenting MiBench [18], which primarily describes the benchmark programs and some aspects of their execution, but does not consider memory bandwidth, we are not aware of other works studying the properties of benchmarks for embedded systems. We thus focus on strategies for memory profiling in the rest of this section. Specifically, we consider three approaches to profiling the memory consumption of applications: hardware counters, simulations, and static analysis.

Hardware Counters. Hardware counters are available on most modern architectures. They require specialized CPU hardware and their implementation is not standardized. Hardware counters achieve high performance and their measures are representative of hardware behaviour. Several projects used hardware counters for profiling multicore systems. Lachaise et al. [20] have developed MemProf, a profiler that allows programmers to choose and implement efficient application-level optimizations for NUMA systems. They rely on an architecture-specific instruction called "ISB" introduced by AMD to perform memory profiling. Traditional profilers, such as Oprofile [3] and Perf [4] are available on ARM architectures, however, they currently do not support our memory controller.

Simulation. Simulators emulate the system architecture in software, allowing performance data to be gathered on any emulated components. To be effective, simulation needs an accurate description of the simulated resources. Another drawback of simulation is the overhead. It is common for a simulator to be 10 to 100 times slower than the native hardware. Valgrind [6] is a widely used instrumentation framework for building dynamic analysis tools. Cachegrind, one of the tools provided by Valgrind, is a cache profiler that provides cache access statistics such as L1/L2 caches misses/hits. The Cachegrind L2 cache implementation uses virtual addresses whereas the SABRE Lite board uses physical addresses.

Static Analysis. Static data-cache analysis techniques rely on the source code and an architecture description. Ghosh et al.'s Cache Miss Equations [17] framework computes tight estimates, but is only applicable to programs that contain regular accesses with predictable patterns. Pellizzoni et al. [24]. propose a time-sliced architecture in which applications must be re-structured into phases that have particular memory-access properties. Boniol et al. [14] propose an algorithm relying on a static analyser [11] for restructuring applications automatically to fit the requirements of the time-sliced architecture.

6 Conclusion

In this paper, we have developed a memory access profiler that relies on hardware counters to measure the global memory traffic issued from applications. We have shown that this profiler has little impact on the behaviour of the profiled applications. Using our profiler, we have traced the memory profiles of the 13 MiBench applications that we have previously found to be strongly affected by the memory usage of applications running on other cores. The resulting memory profiles show that the executions of more than half of these applications involve frequent high memory spikes. To identify the origins of these spikes, we have used a methodology that links the application source code to the memory profile. Based on the results, we have classified the memory spikes into those that are derived from the behaviour of the C standard library, the operating system, and the application. We have used various techniques, including rewriting the application, overloading the C standard library, changing the size of the file system buffers, and recompiling the operating system kernel to isolate the reasons for the observed memory spikes. We have established that C standard Library spikes come from buffered I/O functions and from low level I/O functions. Buffered I/O functions generate memory spikes when they refill their internal buffer, while low level I/O functions produce memory spikes on each access to a memory mapped file. We have shown that operating system spikes are due to the system timer. Finally, we explore some reasons for application spikes with a detailed study of `Susan large c`.

In future work, we plan to develop strategies for recoding the MiBench applications to eliminate the sources of memory bandwidth that derive from the

C standard library or the operating system, so that the MiBench applications better mirror the behaviour of embedded applications.

References

1. Nxp boards. `http://www.nxp.com/`.
2. Okl4 microvisor. `http://www.ok-labs.com/products/okl4-microvisor`.
3. OProfile - a system profiler for Linux. `http://oprofile.sourceforge.net`.
4. perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org`.
5. PikeOS. `http://www.sysgo.com`.
6. Valgrind. `http://valgrind.org`.
7. ARM. *ARM Architecture Reference Manual ARMv7-A|R*, rev C.b, Nov 2012.
8. ARM. *Cortex-A9 Technical Reference Manual*, rev r4p1, June 2012.
9. ARM. *Level 2 Cache Controller L2C-310 TRM*, rev r3p3, June 2012.
10. ARM. *Cortex-A9 MPCore Technical Reference Manual*, June rev r4p1, 2012.
11. C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *STEUS*, pages 35–46. Springer, 2010.
12. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.
13. A. Blin, C. Courtaud, J. Sopena, J. Lawall, and G. Muller. Maximizing parallelism without exploding deadlines in a mixed criticality embedded system. Technical Report RR-8838, INRIA, Jan. 2016.
14. F. Boniol, H. Cassé, E. Noulard, and C. Pagetti. Deterministic execution model on COTS hardware. In *ARCS*, pages 98–110. Springer-Verlag, 2012.
15. S. Fisher. Certifying applications in a multi-core environment: The world's first multi-core certification to sil 4. *SYSGO AG*, 2014.
16. S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles*, 2009.
17. S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *TOPLAS*, 21(4), 1999.
18. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, IEEE International Workshop*, pages 3–14, 2001.
19. X. Jean, M. Gatti, D. Faura, L. Pautet, and T. Robert. A software approach for managing shared resources in multicore ima systems. In *DASC*, Oct. 2013.
20. R. Lachaize, B. Lepers, V. Quéma, et al. MemProf: A memory profiler for NUMA multicore systems. In *USENIX Annual Technical Conference*, pages 53–64, 2012.
21. J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *EDCC*, pages 132–143, May.
22. S. NXP. *i.MX 6Dual/6Quad Processor Reference Manual*, rev 1, April 2013.
23. R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz. From a federated to an integrated automotive architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):956, 2009.
24. R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *RTAS*, 2011.
25. S. Peter, A. Baumann, T. Roscoe, P. Barham, and R. Isaacs. 30 seconds is not enough!: a study of operating system timer usage. In *EuroSys*, 2008.
26. H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS, 2013 IEEE 19th*, pages 55–64. IEEE, 2013.